TIDDLE:

A TRACE DESCRIPTION LANGUAGE FOR GENERATING CONCURRENT BENCHMARKS TO TEST DYNAMIC ANALYSES

CAITLIN SADOWSKI
JAEHEON YI

JULY 20, 2009

UNIVERSITY OF CALIFORNIA AT SANTA CRUZ

SOME DYNAMIC ANALYSES FOR CONCURRENCY

- Data race

Happens Before
Eraser
DJIT+
Goldilocks
FastTrack

- Atomicity

Block-Based
Velodrome
SideTrack
Atomizer
Commit-Node
Atom Fuzzer

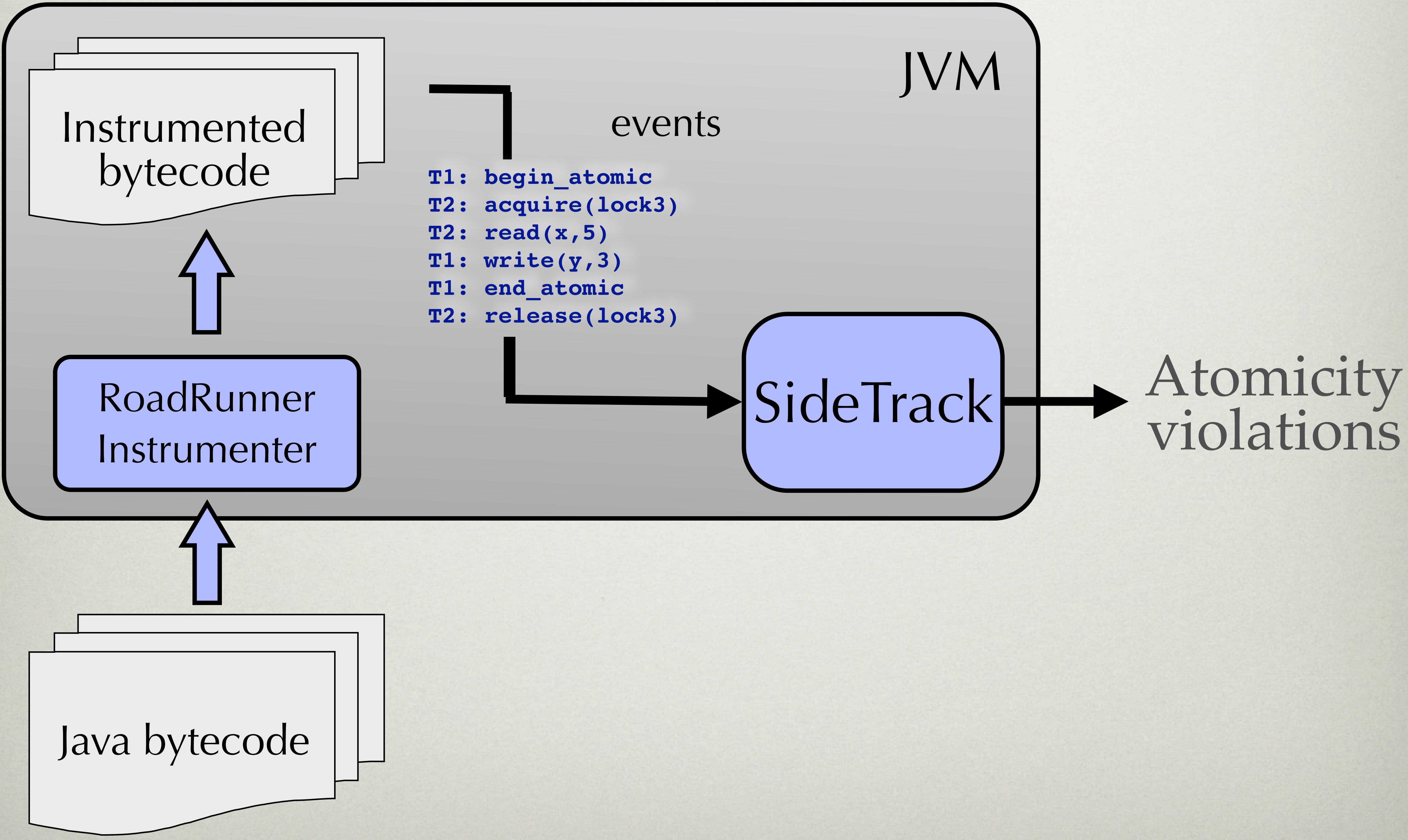
- Deterministic Parallelism

SingleTrack
Burnim09

- Deadlock

GoodLock
Pulse
Agarwal06
Deadlock Fuzzer

RoadRunner



JVM

Instrumented bytecode

events

```
T1: begin_atomic
T2: acquire(lock3)
T2: read(x,5)
T1: write(y,3)
T1: end_atomic
T2: release(lock3)
```

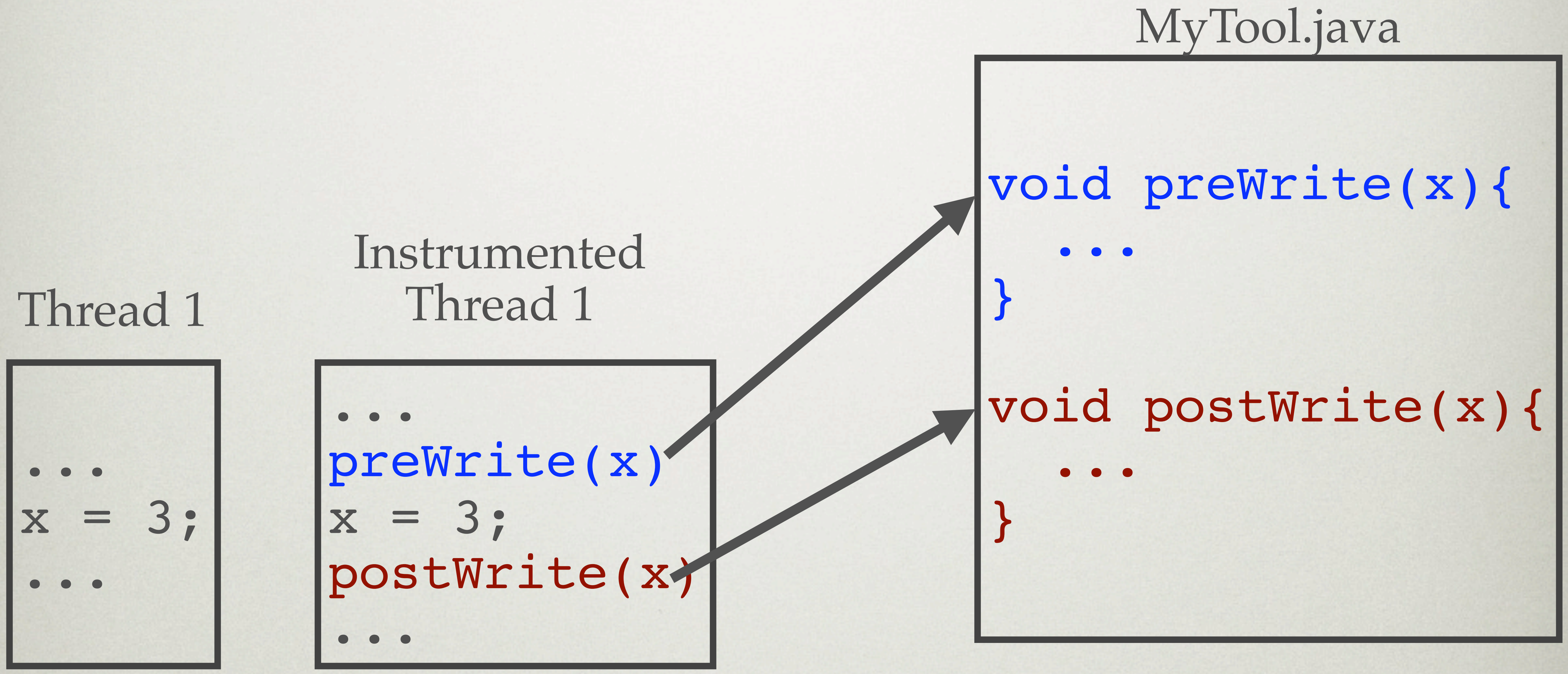
RoadRunner Instrumenter

SideTrack

Atomicity violations

Java bytecode

RoadRunner



Thread 1

```
...
x = 3;
...
```

Instrumented
Thread 1

```
...
preWrite(x)
x = 3;
postWrite(x)
...
```

MyTool.java

```
void preWrite(x){
   ...
}

void postWrite(x){
   ...
}
```

Happens-Before (HB) Race

Thread 1 Thread 2

`write x`

`write x`

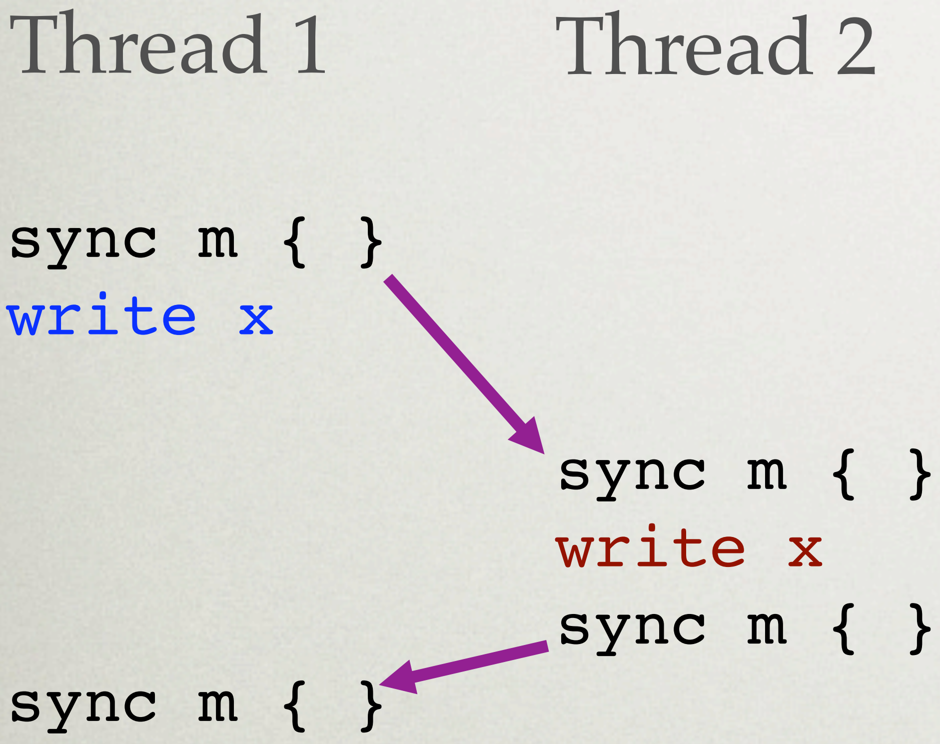
```
Thread t1 = new Thread(){
               void run(){
                 x = 1;
             }};

Thread t2 = new Thread(){
               void run(){
                 x = 2;
             }};

static int x;

public static void main(){
  t1.start();
  t2.start();
}
```

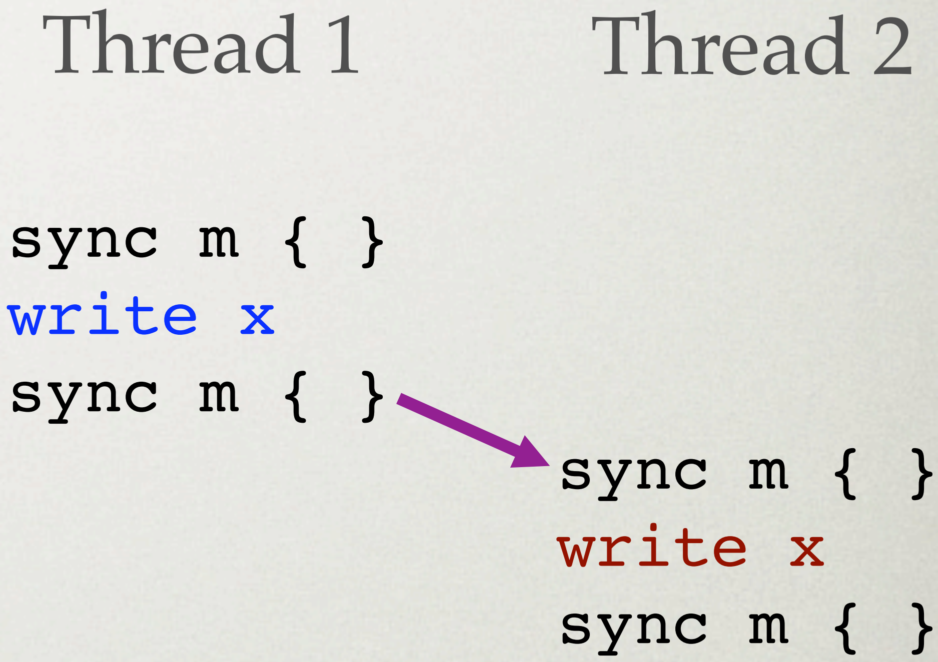
HB Race



```
Thread 1          Thread 2

sync m { }
write x
                  sync m { }
                  write x
                  sync m { }
sync m { }
```

No HB Race

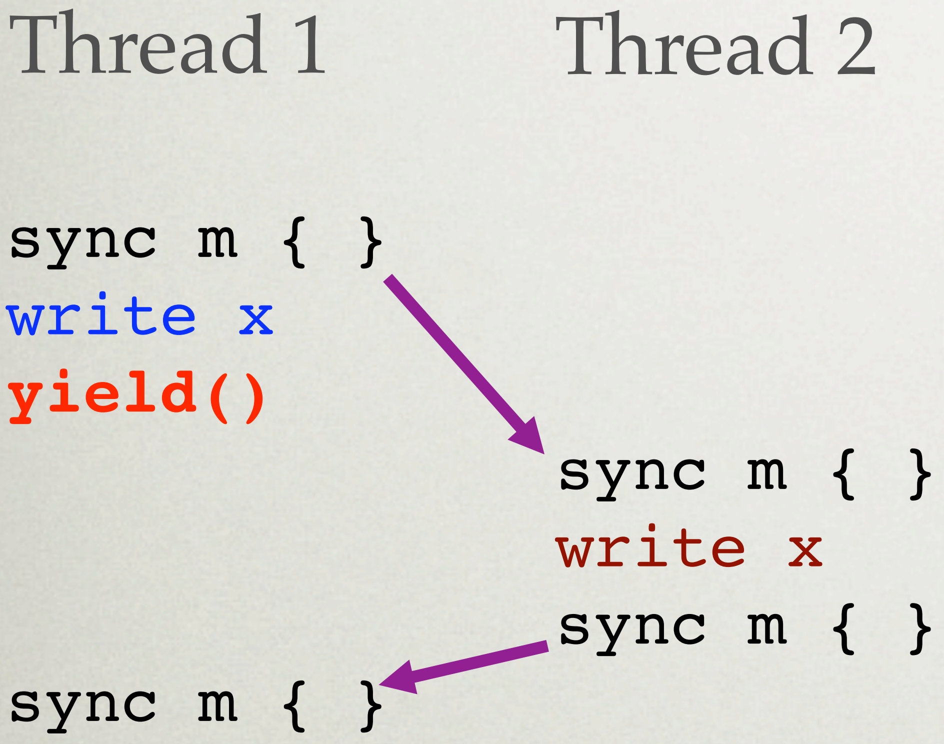


```
Thread 1          Thread 2

sync m { }
write x
sync m { }
                  sync m { }
                  write x
                  sync m { }
```

WHAT ABOUT ADDING YIELDS?

HB Race



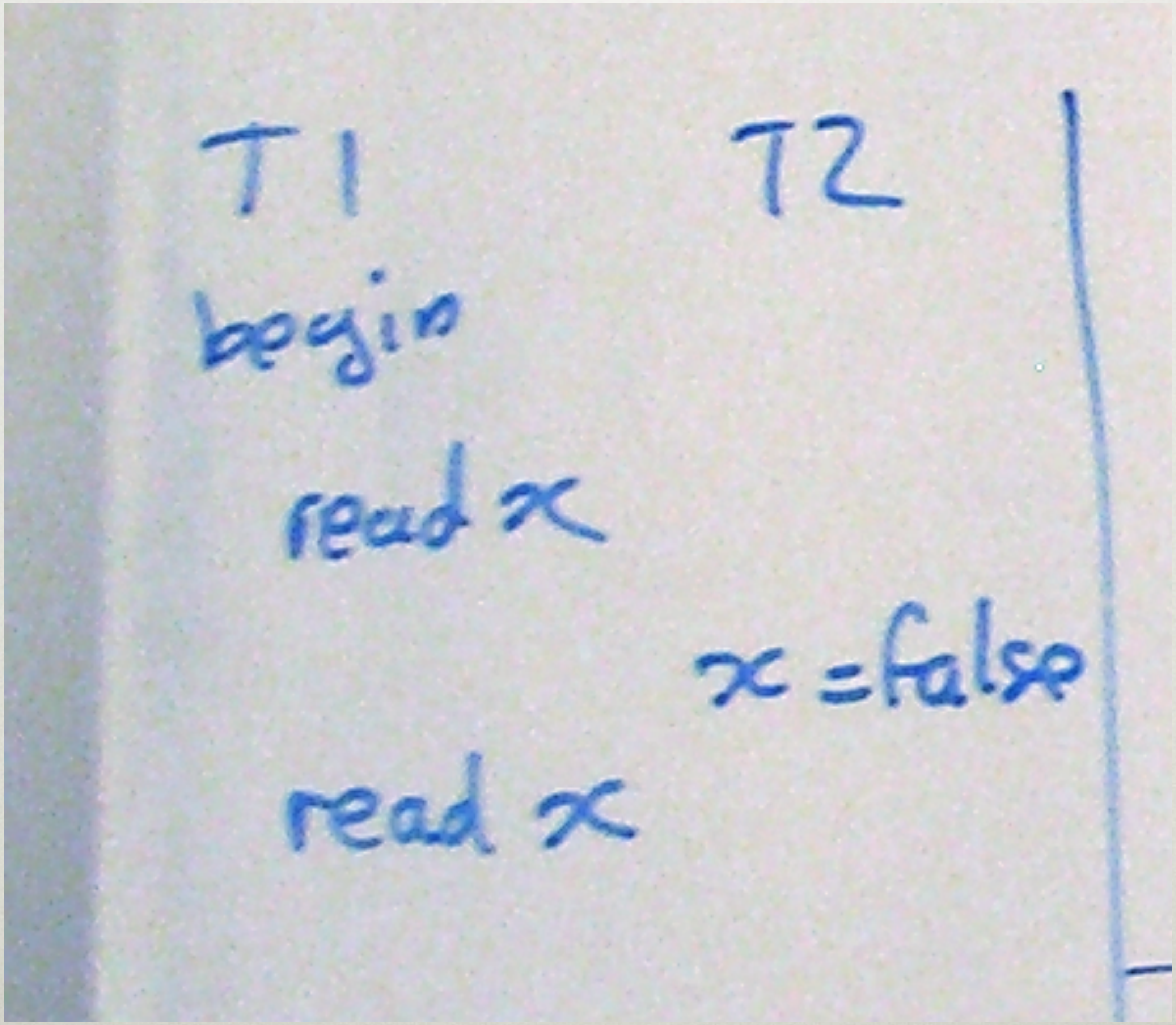
Thread 1 Thread 2

```
sync m { }
write x
yield()
              sync m { }
              write x
              sync m { }
sync m { }
```

- ad hoc
- no guarantees
- can be complex
- difficult to maintain

... SO HOW DO WE WRITE TRACES TO TEST TOOLS?

THINKING UP TEST CASES



T1	T2
begin	
read x	
	x = false
read x	

TIDDLE: A DOMAIN SPECIFIC LANGUAGE FOR DESCRIBING TRACES

```
trace ::= op*
op ::= rd   Tid Var (Val)
     | wr   Tid Var (Val)
     | acq  Tid Lock
     | rel  Tid Lock
     | fork Tid Tid
     | join Tid Tid
     | beg  Tid Label
     | end  Tid Label
Tid ::= Int
Var ::= String
Val ::= Int
Label ::= String
```

QUICK WAY OF TESTING DYNAMIC ANALYSIS

- Race condition:

```
acq 1 m
wr  1 x
rel 1 m
wr  2 x
```

```
//Generated by Tiddle, UC Santa Cruz

package feasibleTests;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class RaceCondition {
  static int x = 0;
  static CyclicBarrier cb = new CyclicBarrier(2);
  static CyclicBarrier cc = new CyclicBarrier(2);
  static int counter = 0;
  static int numThreads = 2;

  static public void await(CyclicBarrier c) throws BrokenBarrierException, InterruptedException {
    c.await();
  }

  static int getCounter() {
    return counter++;
  }


  public static void main(String[] args) {
    final Thread t2 = new Thread() {
      public void run() {
        try {
          int _z = 0;
          await(cc);
          await(cb);
          await(cc);
          x = getCounter();
          await(cb);
        }
        catch (InterruptedException e) {
          e.printStackTrace();
        }
        catch (BrokenBarrierException e) {
          e.printStackTrace();
        }
      }
    };
    final Thread t1 = new Thread() {
      public void run() {
        try {
          int _z = 0;
          await(cc);
          _z = x;
          await(cb);
          await(cc);
          await(cb);
        }
        catch (InterruptedException e) {
          e.printStackTrace();
        }
        catch (BrokenBarrierException e) {
          e.printStackTrace();
        }
      }
    };
    t1.start();
    t2.start();
  }
}
```

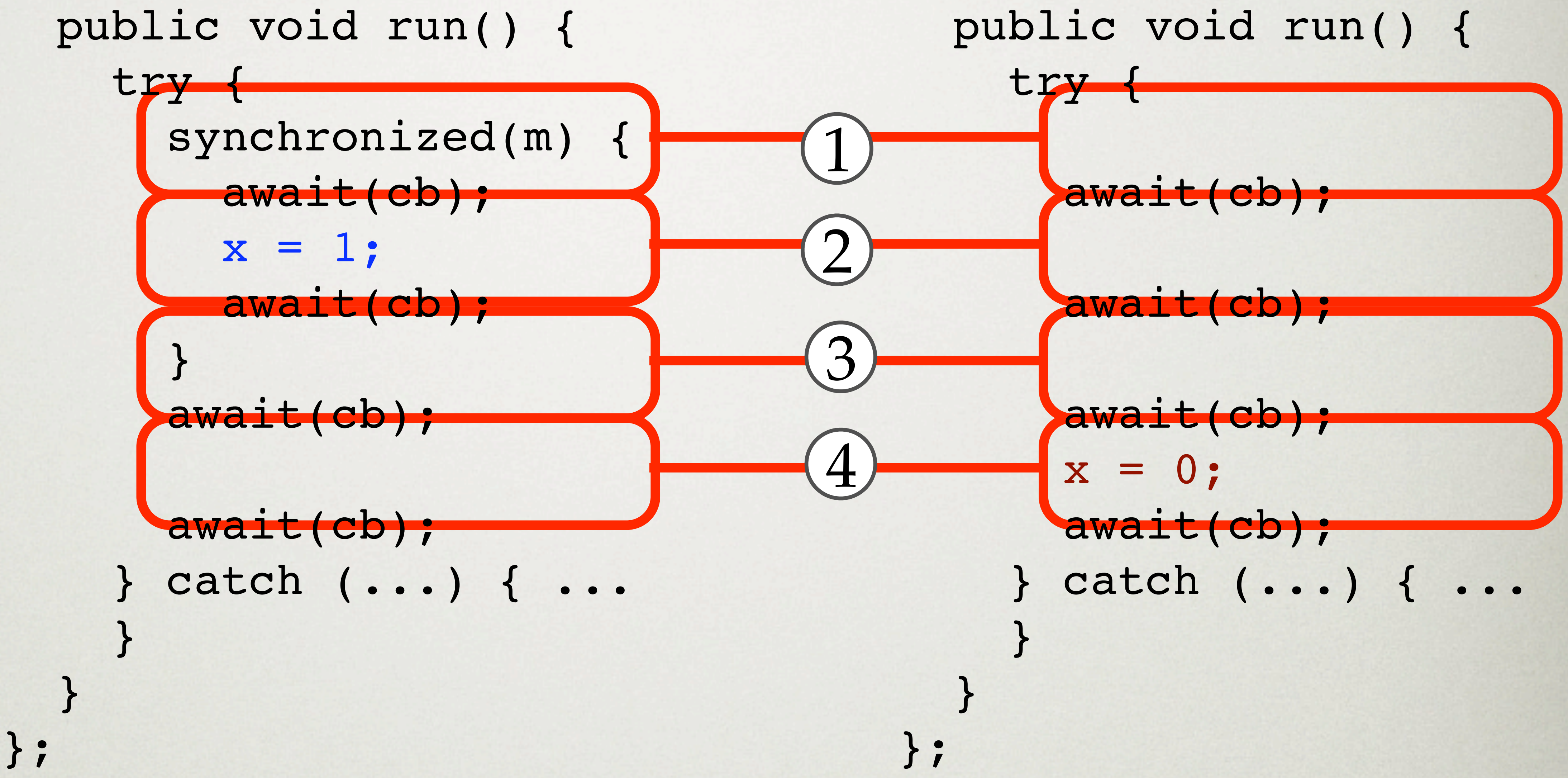
```
acq 1 m
wr  1 x
rel 1 m
wr  2 x
```

```
public class SimpleRace {
  static int x = 0;
  static Object m = new Object();
  static CyclicBarrier cb = new CyclicBarrier(2);

  static public void await(CyclicBarrier c)
    throws BrokenBarrierException, InterruptedException {
    c.await();
  }

  public static void main(String[] args) {
    final Thread t1 = new Thread() {
      public void run() { ...
      }
    };
    final Thread t2 = new Thread() {
      public void run() { ...
      }
    };
    t1.start();
    t2.start();
  }
}
```

```
acq 1 m ①
wr  1 x ②
rel 1 m ③
wr  2 x ④
```



Thread 1

```
 public void run() {
  try {
   synchronized(m) {
    await(cb);
    x = 1;
    await(cb);
   }
   await(cb);

   await(cb);
  } catch (...) { ...
  }
 }
};
```

Thread 2

```
 public void run() {
  try {

   await(cb);

   await(cb);

   await(cb);
   x = 0;
   await(cb);
  } catch (...) { ...
  }
 }
};
```

UNIT TESTING FOR DYNAMIC ANALYSIS

- Easy to describe test case as a trace
- Compile trace to multithreaded test
  - barriers for determinism
  - avoid boilerplate and complexity

Some Dynamic Analyses for Concurrency

- Data race
  - Happens Before
  - Eraser
  - DJIT+
  - Goldilocks
  - FastTrack
- Atomicity
  - Atomizer
  - Block-Based
  - Commit-Node
  - Velodrome
  - Atom Fuzzer
  - SideTrack
- Deterministic Parallelism
  - SingleTrack
  - Burnim09
- Deadlock
  - GoodLock
  - Pulse
  - Agarwal06
  - Deadlock Fuzzer

ATOMICITY

The effect of an atomic code block can be considered in isolation from the rest of a running program.

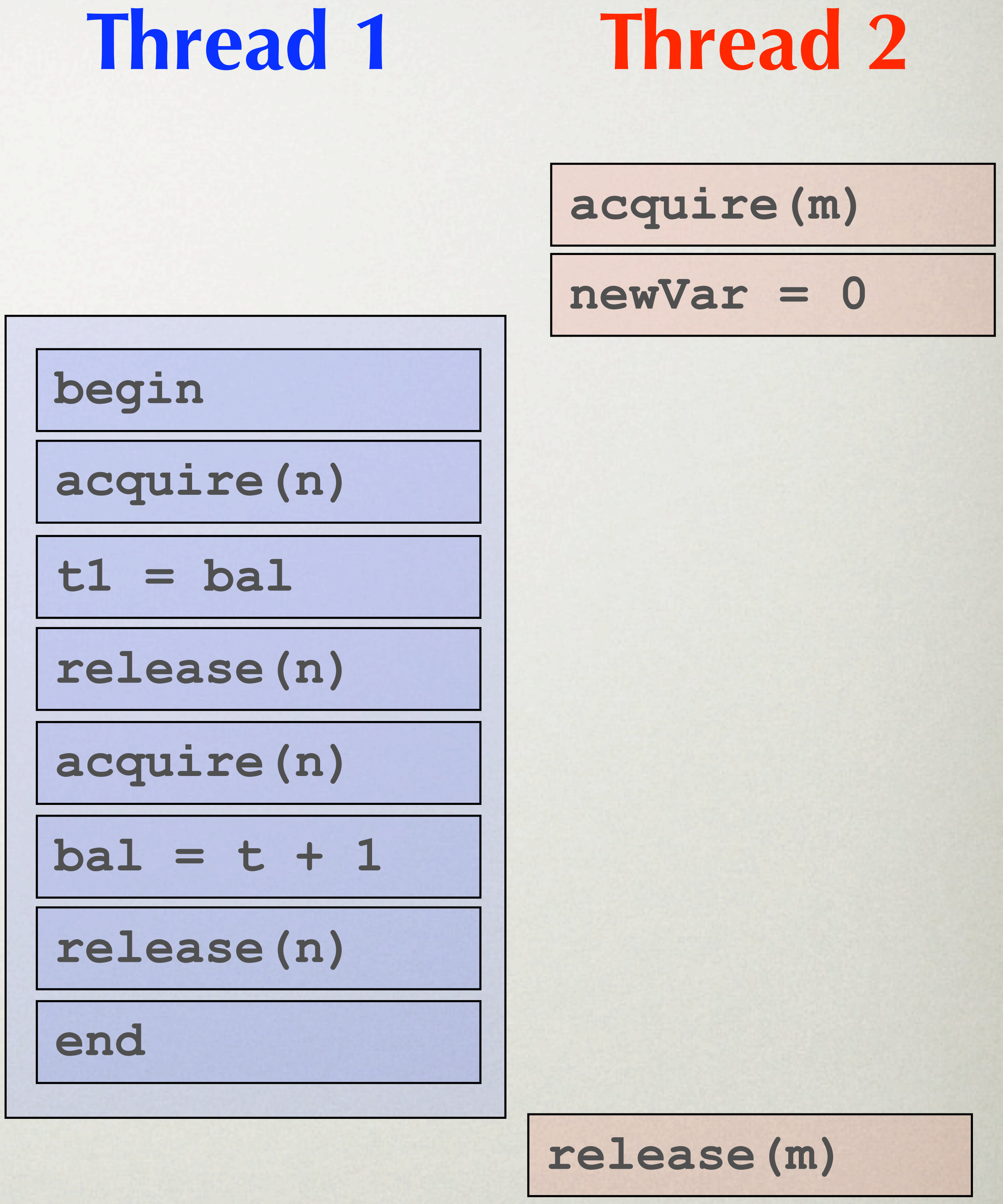
- enables sequential reasoning
- atomicity violations often represent synchronization errors
- most methods are atomic

Thread 1

```
atomic{
  synchronized(n){
      tmp = bal;
  }
  synchronized(n){
      bal = tmp + 1;
  }
}
```

Thread 2

```
  synchronized(m){
     newVar = 0;
  }
```



Thread 1

Thread 2

`acquire(m)`

`newVar = 0`

`begin`

`acquire(n)`

`t1 = bal`

`release(n)`

`acquire(n)`

`bal = t + 1`

`release(n)`

`end`

`release(m)`

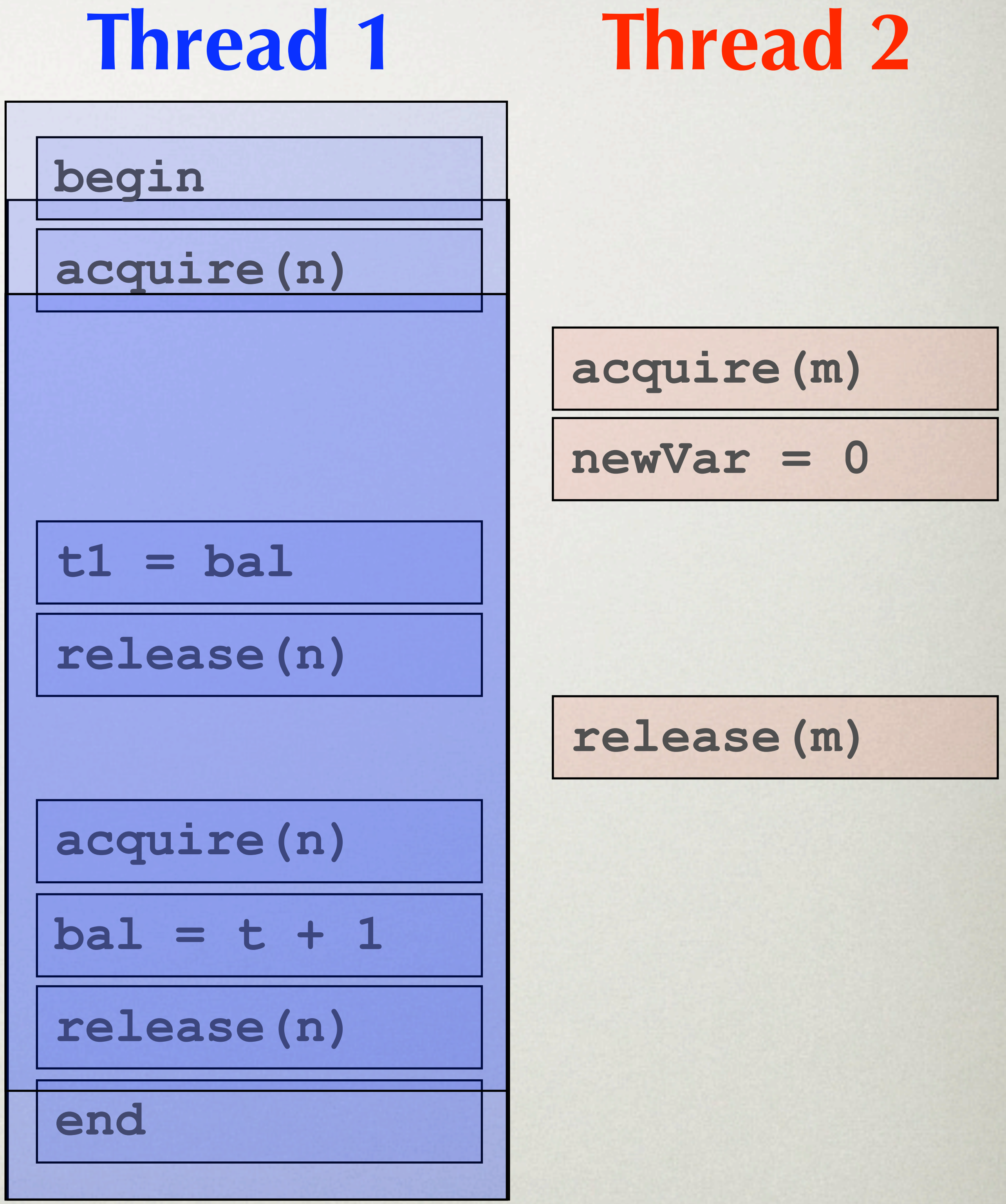
Serial Trace: Each atomic block executes contiguously

Thread 1

```
atomic{
  synchronized(n){
      tmp = bal;
  }
  synchronized(n){
      bal = tmp + 1;
  }
}
```

Thread 2

```
  synchronized(m){
     newVar = 0;
  }
```



Thread 1

Thread 2

```
begin
acquire(n)
                        acquire(m)
                        newVar = 0
t1 = bal
release(n)
                        release(m)
acquire(n)
bal = t + 1
release(n)
end
```

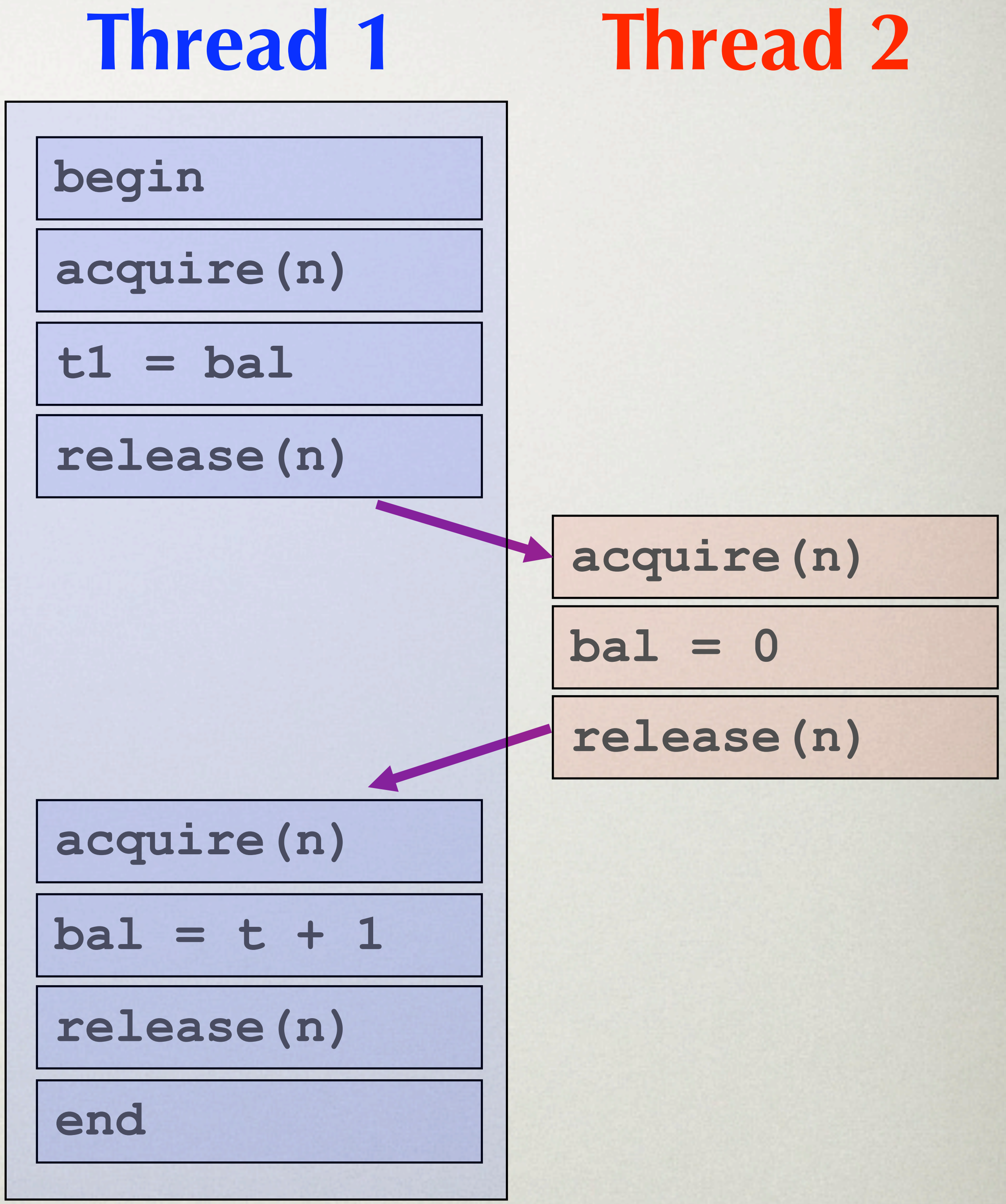
Atomicity = Serializability

Thread 1

```
atomic{
  synchronized(n){
      tmp = bal;
  }
  synchronized(n){
      bal = tmp + 1;
  }
}
```

Thread 2

```
  synchronized(n){
    bal = 0;
  }
```



Thread 1

Thread 2

```
begin
acquire(n)
t1 = bal
release(n)
```

```
acquire(n)
bal = 0
release(n)
```

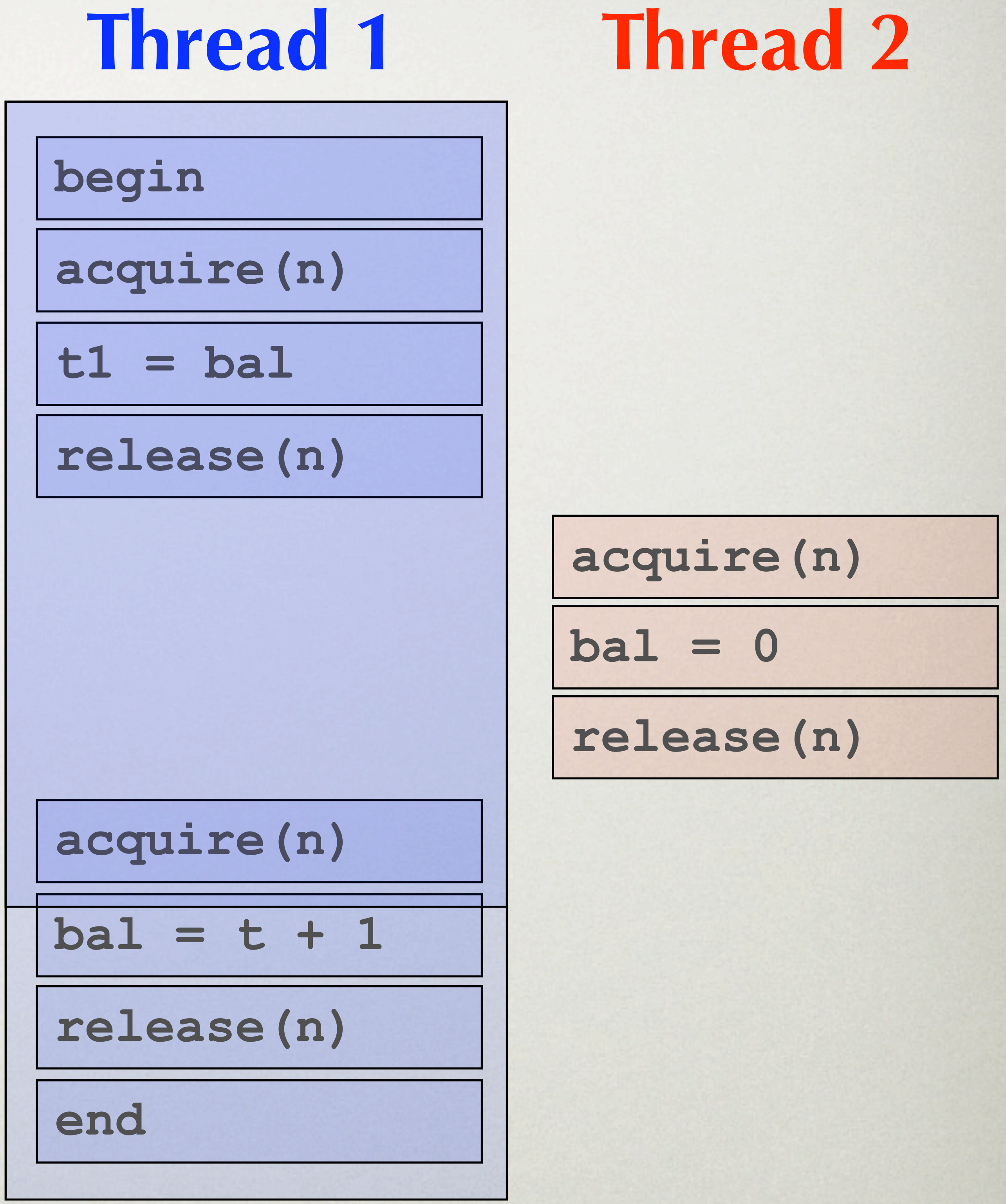
```
acquire(n)
bal = t + 1
release(n)
end
```

Thread 1

```
atomic{
  synchronized(n){
      tmp = bal;
  }
  synchronized(n){
      bal = tmp + 1;
  }
}
```

Thread 2

```
  synchronized(n){
    bal = 0;
  }
```



Thread 1

Thread 2

```
begin
acquire(n)
t1 = bal
release(n)
                          acquire(n)
                          bal = 0
                          release(n)
acquire(n)
bal = t + 1
release(n)
end
```

QUICK WAY OF TESTING DYNAMIC ANALYSIS

- HB Atomicity Violation:

```
beg 1 b
wr  1 x
wr  2 x
rd  1 x
end 1 b
```

```java
//Generated by Tiddle, UC Santa Cruz

package feasibleTests;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class AtomicityViolation {
  static int x = 0;
  static CyclicBarrier cb = new CyclicBarrier(2);
  static CyclicBarrier cc = new CyclicBarrier(2);
  static int counter = 0;
  static int numThreads = 2;

  static public void await(CyclicBarrier c) throws BrokenBarrierException, InterruptedException {
    c.await();
  }

  static int getCounter() {
    return counter++;
  }

  static void b() throws BrokenBarrierException, InterruptedException  {
    await(cb);
    await(cc);
    x = getCounter();
    await(cb);
    await(cc);
    await(cb);
    await(cc);
    _z = x;
    await(cb);
    await(cc);
  }

  public static void main(String[] args) {
    final Thread t2 = new Thread() {
      public void run() {
        try {
          int _z = 0;
          await(cc);
          await(cb);
          await(cc);
          await(cb);
          await(cc);
          x = getCounter();
          await(cb);
          await(cc);
          await(cb);
          await(cc);
          await(cb);
        }
        catch (InterruptedException e) {
          e.printStackTrace();
        }
        catch (BrokenBarrierException e) {
          e.printStackTrace();
        }
      }
    };
    final Thread t1 = new Thread() {
      public void run() {
        try {
          int _z = 0;
          await(cc);
          b();
          await(cb);
        }
        catch (InterruptedException e) {
          e.printStackTrace();
        }
        catch (BrokenBarrierException e) {
          e.printStackTrace();
        }
      }
    };
    t1.start();
    t2.start();
  }
}
```

QUICK WAY OF TESTING DYNAMIC ANALYSIS

- HB Atomicity Violation:

```
beg 1 b
acq 1 m
rel 1 m
acq 2 m
rel 2 m
acq 1 m
rel 1 m
end 1 b
```

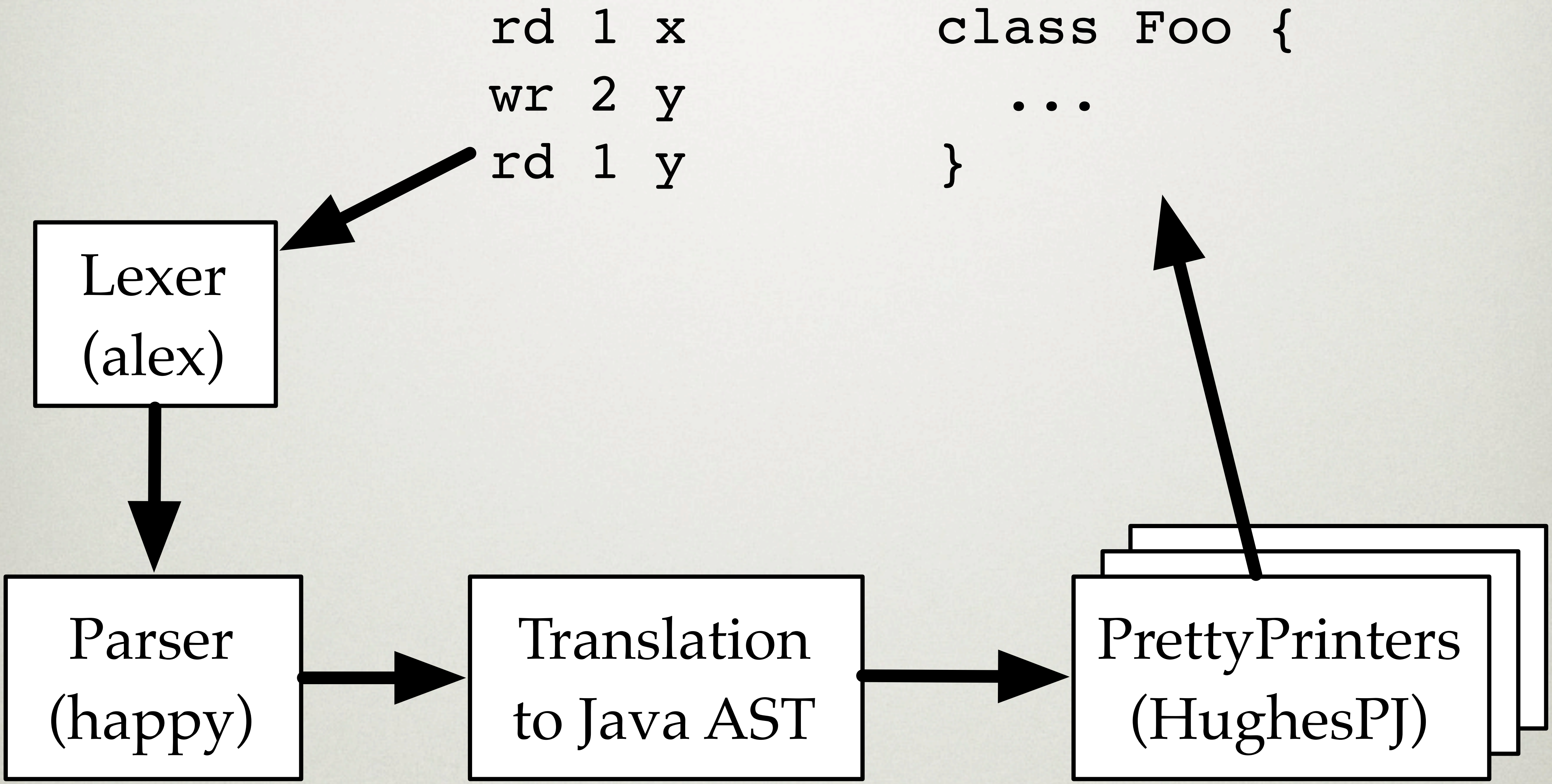
QUICK WAY OF TESTING DYNAMIC ANALYSIS

- Serial trace
- Feasibly non-serializable

```
beg 1 b
acq 1 m
rel 1 m
```

```
acq 1 m
rel 1 m
end 1 b
acq 2 m
rel 2 m
```

TIDDLE COMPILER



```
rd 1 x
wr 2 y
rd 1 y
```

```
class Foo {
  ...
}
```

Lexer
(alex)

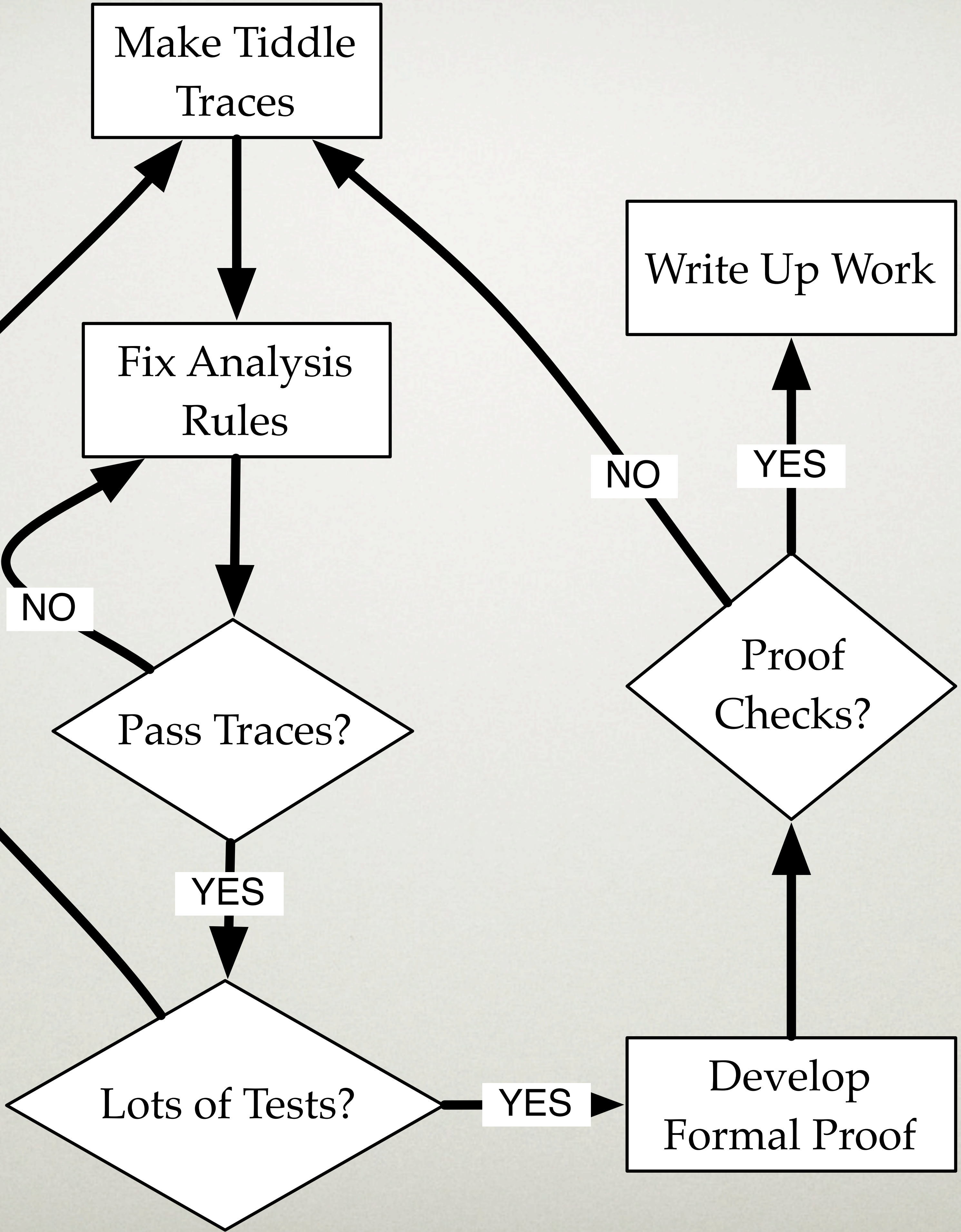
Parser
(happy)

Translation
to Java AST

PrettyPrinters
(HughesPJ)

EQUIVALENT TRACES

- Equivalent modulo happens-before
  - Different total orderings of same HB partial ordering
- Analyses should behave same way
  - Regression testing for analyses



Make Tiddle Traces

Fix Analysis Rules

Pass Traces?

NO

YES

Lots of Tests?

NO

YES

Develop Formal Proof

Proof Checks?

NO

YES

Write Up Work

NEXT STEPS

- Support for other Java features
  - volatiles, wait/notify, etc.
- Bug capture
- Specification language

NOT A PANACEA

- Helps you with the *known* unknowns
- This won’t help you find the bugs you’re not expecting
  - Doesn’t help you with the *unknown* unknowns

CONCLUSION

- Domain-specific language
  - describe bugs succinctly
- Generate Java test cases
  - multithreaded, deterministic
- Helps dynamic analysis development
  - 100+ Tiddle traces to date